CISC 322/326 Assignment 2

Concrete Architecture of Kodi

Group 19 – Mystery Inc.

November 19, 2023

Authors

Brian Cabingan | 20bjc9@queensu.ca
Ethan Fighel | 20eaf4@queensu.ca
Zachary Kizell | 20zk16@queensu.ca
Daniel Madan | 18djm10@queensu.ca
Brett Morris | 19bcm2@queensu.ca
Daniel Solomon | 21ds28@queensu.ca

Table of Contents

Abstract

This paper conducts an in-depth exploration of Kodi's concrete architecture and its unconventional deviations from conventional models attributable to its open-source character, and the intricate layers of its architectural style, along with a detailed examination of each layer's functionalities. It further employs a reflection analysis to illuminate the disparities between the theoretical and practical architectural manifestations, underscoring the influence of Kodi's open-source principles. In addition to this, practical user scenarios are presented to vividly illustrate interactions with subsystem features, including the creation of bookmarks and the selection of subtitles. The paper also includes comprehensive discussions on essential data dictionaries and naming conventions. Finally, it encapsulates the key lessons derived from this comprehensive analysis.

Introduction & Overview

Kodi's concrete architecture, which extends from its conceptual architecture, follows a layered architecture, comprising four distinct layers: the Client Layer, Presentation Layer, Business Layer, and Data Layer. To derive this concrete architecture, our team analyzed various sources, including the GitHub repository, Understand for dependency analysis, and the official Kodi Wiki. We used Understand's dependency and graphing capabilities to visualize component connections and delved into the source code to understand specific interactions and component roles. Kodi's layered architecture style differs from traditional models due to its open-source nature, with key subsystems identified, such as the User Input Subsystem, GUI Subsystem, and Player Core Subsystem. The reflexion analysis reveals discrepancies between the conceptual and concrete architecture, attributed to Kodi's open-source nature, with notable deviations such as direct database calls for basic video information and unconventional subtitle functionality through add-ons.

Concrete Architecture

Derivation Process

To derive the concrete architecture of Kodi, our team analyzed a number of relevant sources. We examined the source code from the GitHub repository and dependencies between subsystems with Understand and used these analyses along with the official Kodi Wiki to build the concrete architecture for the system. The dependency and graphing capabilities of Understand helped us visualize the connections of various components within the system, and delving into the raw source code proved valuable for gaining insights into specific interactions and understanding the roles of each component.

The first step we took in building an architecture in Understand was looking at the folder and file names in the source code. Some of the file names we could organize into the architecture just off of intuition. Other files had names that did not clearly fit into a particular system or subsystem; that is we had to look at the code itself to determine where the file should go. Sometimes even the source code was not commented enough to clearly fit into one subsystem, so we found it helpful to look at the dependencies and interactions of just a single file and then see where the files it interacted with fit into our architecture. We referenced the architecture page on the Kodi Wiki quite often, as it gave us a general idea of what subsystems we should be forming within each layer along the way.

We discovered Kodi has a Layered Architecture style that is split into four layers: Client Layer, Presentation Layer, Business Layer, and Data Layer. As described on the Kodi Wiki, we found that our system did fit into a layered architecture style, but not in a traditional way like how we learned in CISC 322 lectures. As we learned in class, typical layered architectures function where only adjacent layers communicate with one another, however in the architecture that we built, it seemed as if no matter how we organized it, non-adjacent layers would always have a large number of interactions between one another. We believe this is largely due to the fact that Kodi is an open-source project with hundreds of contributors, so there would be a large variance of coding backgrounds between the many contributors. It is also less likely for open-source projects to follow a strict architecture, so it is not surprising that the layered architecture of Kodi is not a traditional one.

Architecture Overview

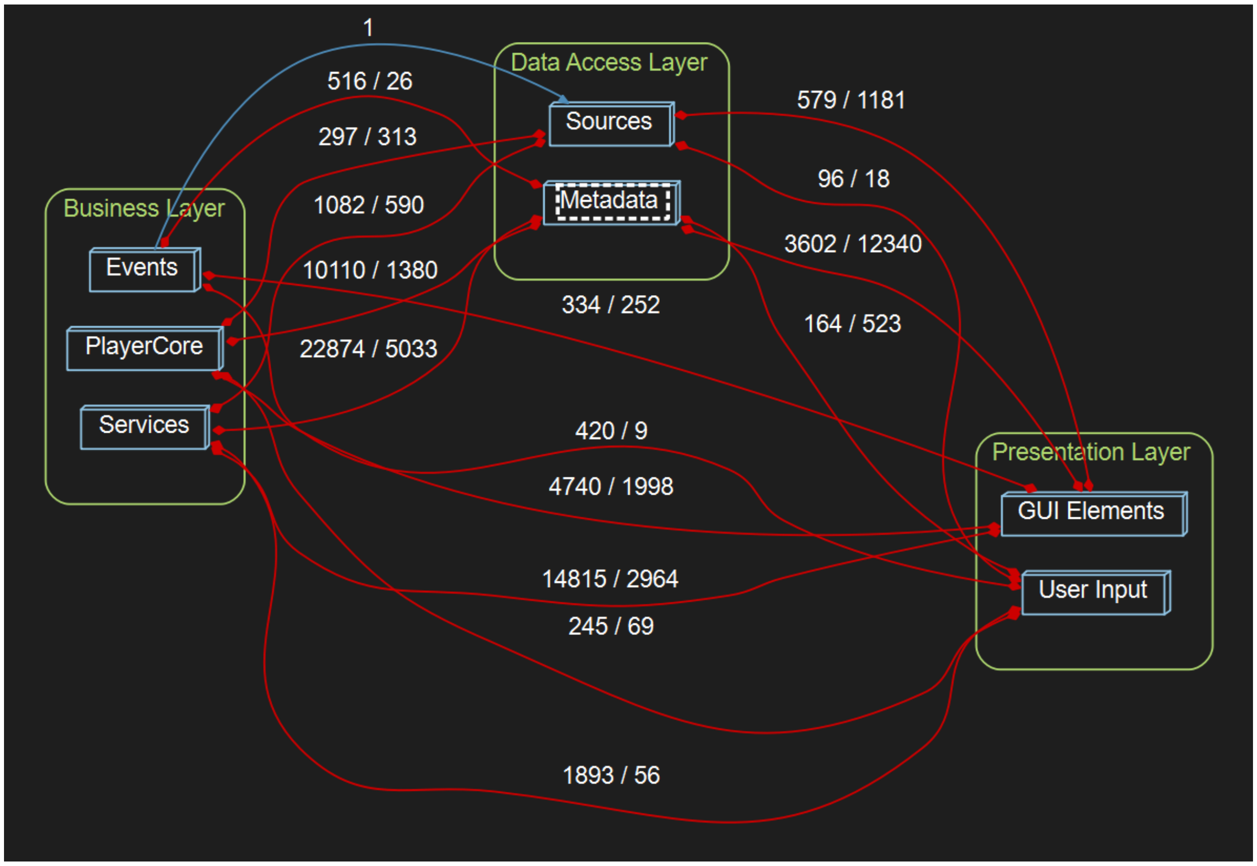

Figure 1. Concrete Architecture Diagram of Kodi (from Understand)

Presentation Layer:

User Input Subsystem

The User Input Subsystem is responsible for capturing all of the user's input and communicating with various GUI elements and menus, as well as for the various games the user can play through Kodi. It is equipped to handle input from various hardware such as keyboards and game controllers, and makes use of an event server to accept device input.

GUI Subsystem

The GUI subsystem is used to manage various aspects of the Graphical User Interfaces that are visible to users. It is used to control changes in the color, font, and size of various GUI elements as the user interacts with them. It is also used to manage window size (fullscreen, windowed,etc.). While user interaction with the GUI is made simple, there are files for every type of element (radio buttons, text elements, etc.) with defined rules and functions that help make the user experience smooth. This subsystem works with systems all throughout Kodi, including the addons, event and games modules to display information to the user.

Business Layer:

Player Core Subsystem

The Player Core Subsystem is responsible for rendering video and audio for the user. It is capable of rendering video from a variety of sources, including the computer's DVD player and video files the user has downloaded. This subsystem is also capable of playing retro video games that the user has the disc for. Kodi makes use of its own in-house developed audio-player, called the PAPlayer, which makes use of codecs to play audio. All of this is relayed to the GUI subsystem, and they work together to display video and information to the user. This subsystem communicates with the input, interfaces and web server subsystems as parts of the process for displaying video and audio for the user.

Python Subsystem

This subsystem is responsible for managing add-ons that the user can add to alter their experience using Kodi. While the subsystem is largely coded in C++, add-ons are coded in Python and can be made by anyone. This system has scripts to handle various parts of the add-ons, such as installers, image decoders, new scrapers, and more. These plug-ins can be downloaded by the user by connecting to the various repos that contain the add-on code.

Web Server Subsystem

This subsystem allows the user to manage their content remotely, either through their web browser or from other applications. Multiple users can access the same content if they are connected to a common local network. This subsystem deals with events and communicates with an event server to process user requests. It also communicates with the GUI subsystem to get user information such as username and password for authentification.

Data Access Layer:

Sources Subsystem

In the sources subsystem of the data access layer, the focus is on effectively managing a variety of media sources. The main goal is to provide users with a simple way to access content from different sources like local storage drives (HDDs, CDs, DVDs) and various network protocols (HTTP, FTP, SMB, etc.). Many files in this subsystem are related to network access and settings, along with settings for handling different types of media sources. As described above, this layer plays a crucial role in enhancing accessibility to different sources of media over the network.

Metadata Subsystem

The metadata subsystem contains and manages additional information and other important settings of both the media and media players. This subsystem is one of the most important in the architecture of Kodi as many of the files are depended on by all other subsystems. Some of the files (such as ServiceBroker.cpp and Util.cpp) contain important configuration information that must be accessed by many files in all of the other subsystems. This subsystem handles importing and exporting, storing and organizing, and even transformation of metadata, among other things. For these reasons, it is one of the most key subsystems for the proper functioning of Kodi.

Analyzed Subsystems

One of Kodi's main functions is its video playback feature, with its video player on-screen display (OSD) being the main method to allow the user to interact with the video player. The video player OSD can be divided into different parts: the application, bookmarking, dialogues, GUI information, file item, and service broker.

The application part of the OSD consists of two files: Application.cpp, and Application.h. Application.cpp handles most of Kodi's functions, but the ones that are more focused on the video player OSD are retrieving the matching resolution based on GUI settings, updating the window resolution, toggling HDR on and off, and showing the information about the current video. Application.h handles additional player functions such as returning the total and current time in fractional seconds of the currently playing media.

Another feature of the video player OSD is the user's ability to add a bookmark. Bookmark.cpp saves a video along with additional information like the episode number, season number, time in seconds, and part number. Bookmark.h checks if the bookmark has been set and if the bookmark is partway through the video file.

Dialogues are windows that communicate information to the user and prompt them for a response. (Kodi development, 1) GUIDialogSubtitles.cpp retrieves a directory and stores a string as well as extract a specific language and the appropriate extension for that language. GUIDialogVideoBookmarks.cpp opens the database, retrieves the bookmarks for the current movie, and adds items to the file list. GUIDialogVideoInfo.cpp gets the video information tags, compares the file items by the database index, checks if it's resumable, and sets additional information like if it's a movie or show and which season and episode it's on. GUIDialogVideoOSD.cpp checks for movement of the mouse or a submenu open which will show the OSD on an action. GUIDialogVideoSettings.cpp handles different video settings such as: brightness, contrast, gamma and orientation.

The GUI information displays all the relevant information to the user. VideoGUIInfo.cpp gets the labels for the video such as the title, genre, director, cast, subtitle language, and video resolution. GUIMediaWindow.cpp formats file items as well as configures the view control as it enables or disables the states of the controls. PlayerGUIInfo.cpp Includes several key pieces of information for the video player such as the volume, start time, rewind and fast forward speed, and video fps.

Other components of the video player OSD include the file item and the service broker. FileItem.cpp initializes the file item type while FileItem.h checks the item type, checks if the item has a valid resume point set, and returns the current resume time. ServiceBroker.cpp calls on various functions such as GetGUI and GetMediaManager.

Reflexion Analysis

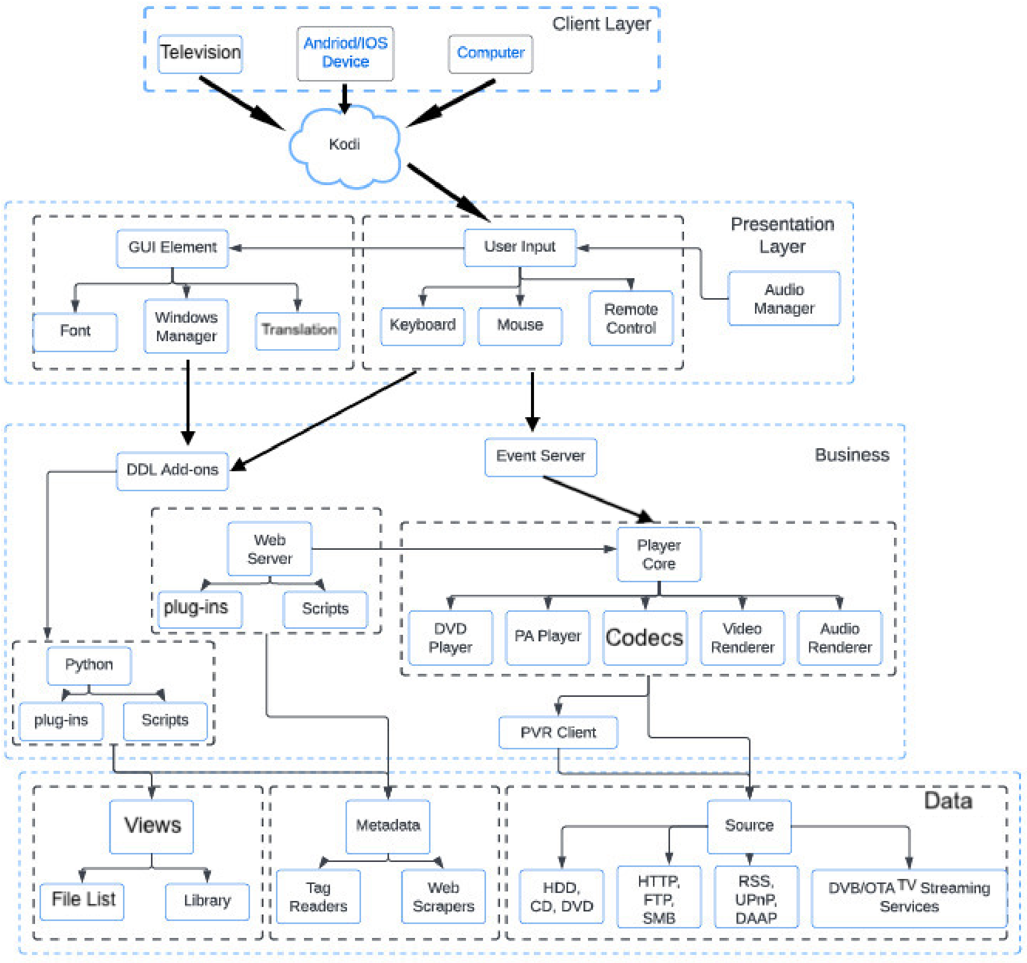

Figure 2. Group 7's Conceptual Architecture (Restyled for more ease of viewing)

High-Level Architecture:

The concrete architecture of Kodi has resulted in a layered style as predicted in the conceptual architecture, but varies from it in terms of what the actual layers are. The upwards interaction between the business layer and the presentation layer is the same, but the presentation layer interacts with the data layer in a similar way to the business layer. In other words, the business layer is not the middleman between the presentation layer and the data layer, which explains why the views elements cannot be found outside of the conceptual architecture.

Note: The underlying reason behind a lot of these discrepancies is this: Kodi is open-source which means that it is built with volunteer programmers creating source patches and bug fixes in their spare time. That results in a lack of awareness, care, or both about the overarching layered style of architecture that is intended for Kodi. This was somewhat articulated earlier in the document as well (Wikipedia Contributors).

These discrepancies are exemplified below in the **unexpected dependencies** found through analysis of the source code:

Presentation Layer: GUI Elements ←→ Business Layer: Events
Rationale: In order to determine the events that occur when interacting with a specific GUI element, the layer must be in direct contact with the event coordination layer to produce a result. Likewise, the GUI elements may be changed depending on current events, such as interactable elements changing their function in specific situations or text may be updated to reflect an updated change.

Presentation Layer: GUI Elements ←→ Business Layer: PlayerCore
Rationale: Toggleable GUI elements such as volume settings, caption settings, volume settings and more would need to be edited in the Presentation Layer and sent to the Business Layer to update the video's settings. The GUI layout receives changes from the PlayerCore element to suit the needs of the media being played, such as timestamps/quality settings for a video or the built-in GUI of a video game.

Presentation Layer: GUI Elements ←→ Data Access Layer: Metadata
Rationale: The information about a specific piece of media stored in the Metadata element will be shared to the GUI Elements to display that information to the user. Conversely, the Metadata depends on triggers from the GUI Elements to know what to send.

Presentation Layer: GUI Elements ←→ Data Access Layer: Sources
Rationale: Methods of accessing specific files such as through storage drives, network protocols and streaming devices will need to be selected by the player, so the GUI elements would need to be updated to reflect these options. It does not work conversely since the source elements do not

await input from GUI since the GUI in this case is the one awaiting input from the source elements and not the other way around.

Presentation Layer: User Input ←→ Data Access Layer: Metadata
Rationale: The Metadata component contains configuration settings that affect how the user is able to interact with the menus, such as enabling touch screen support or controller access for menus. Conversely, the Metadata responds to triggers from the user input to know what to send.

Presentation Layer: User Input ←→ Data Access Layer: Sources
Rationale: When selecting a video or game to play, the user can add a source file to their Sources database by typing in the media file's destination directly. Once the file has been added to the Sources component, it displays the media in its database for the user to select. Conversely, the Sources component responds to triggers from the user input to know what piece of media to grab from the file system.

Business Layer: Events ←→ Data Access Layer: Metadata
Rationale: The event server has to know the attributes of the media file it is interacting with in order to properly configure controls. Conversely, the metadata depends on triggers from the event server to know when and where to send the data.

Business Layer: Events → Data Access Layer: Sources
Rationale: Playing a media file that the user selects is an event that requires access to all of the files that have been previously added to the Sources component.

Presentation Layer: User Input → Business Layer: PlayerCore
Rationale: The PlayerCore component loads and displays the media that the user interacts with, providing visual information on the choices they are making.

Business Layer: PlayerCore → Data Access Layer: Metadata
Rationale: Some aspects of a video or game that is loaded by the PlayerCore component require specific configuration settings stored in the files of the Metadata component.

Business Layer: Events → Data Access Layer: Sources
Rationale: When the user selects a piece of media from the menu, the event begins playing the media using the original file from the Sources database.

Video Player OSD Reflexion Analysis:

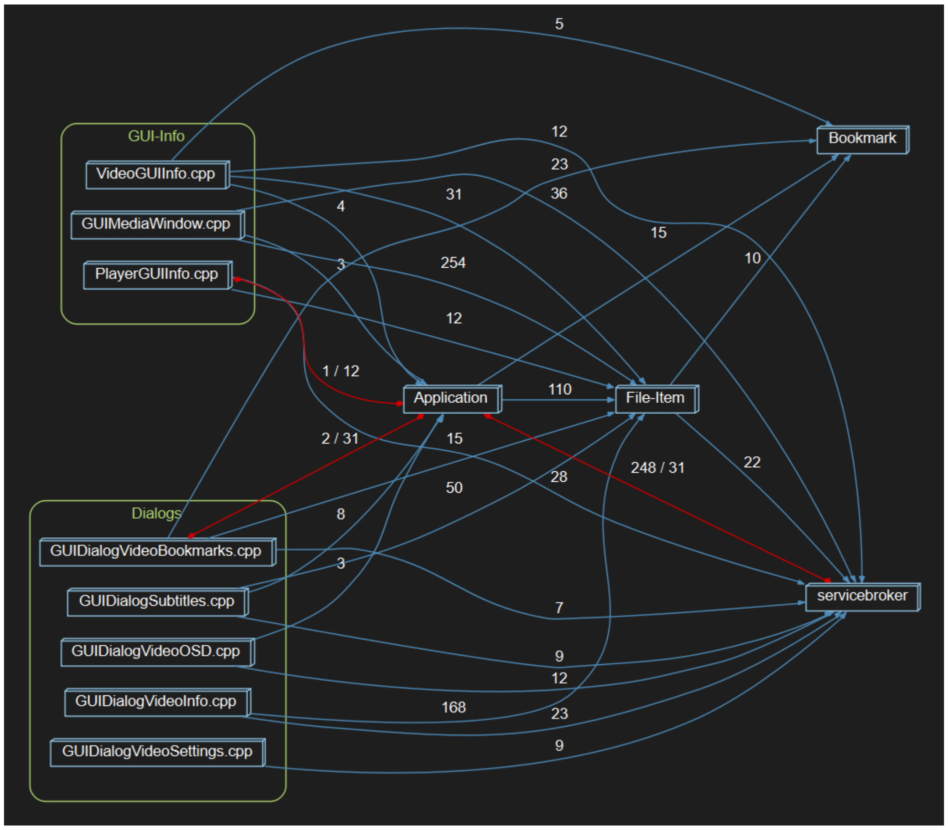

Figure 3. Subsystem Architecture of Video Player OSD

In order to more easily understand the discrepancies between the conceptual and concrete architecture, a reflexion analysis is performed on the video player OSD.

When getting the basic info about the video such as title, genre, duration, and other user-relevant information, it calls on a program that gets it directly from a database. In light of the conceptual architecture, one would assume that the web server would have an infrastructure to mediate calls between the database and a presentation layer component, but here it is nowhere to be found.

Rationale: Since the information about this specific piece of media is stored in the database and can be called directly, it removes a reason for going through unnecessary architectural middlemen in the perspective of a singular volunteer programmer.

Additionally, the subtitles functionality also diverges from the conceptual architecture, but in a different manner than the retrieval of video info. It uses addons to download new subtitles from external sources, which fits the conceptual mold, but it also has the ability to load subtitles directly from the file system that have either been installed from the addons, or just part of the package of the media file that it is meant for.

Rationale: To access subtitles locally, it is most convenient and optimal for the developer of the subtitle component to program a direct access rather than jump through the web server every time because of the ease in which it can be implemented and the perceived efficiency from a singular volunteer programmer's point of view.

Use Cases:

Use case 1: Choosing Subtitles for a newly downloaded video file

To access subtitles for a video, follow these steps: First, open the video player's On-Screen Display (OSD). Then, select the subtitles widget, which will display a settings dialogue. You have two options for subtitles: either browse through your file system to choose one, or download them from the add-on provider specifically for that media. Once you've made your selection, press 'close' to save the changes.

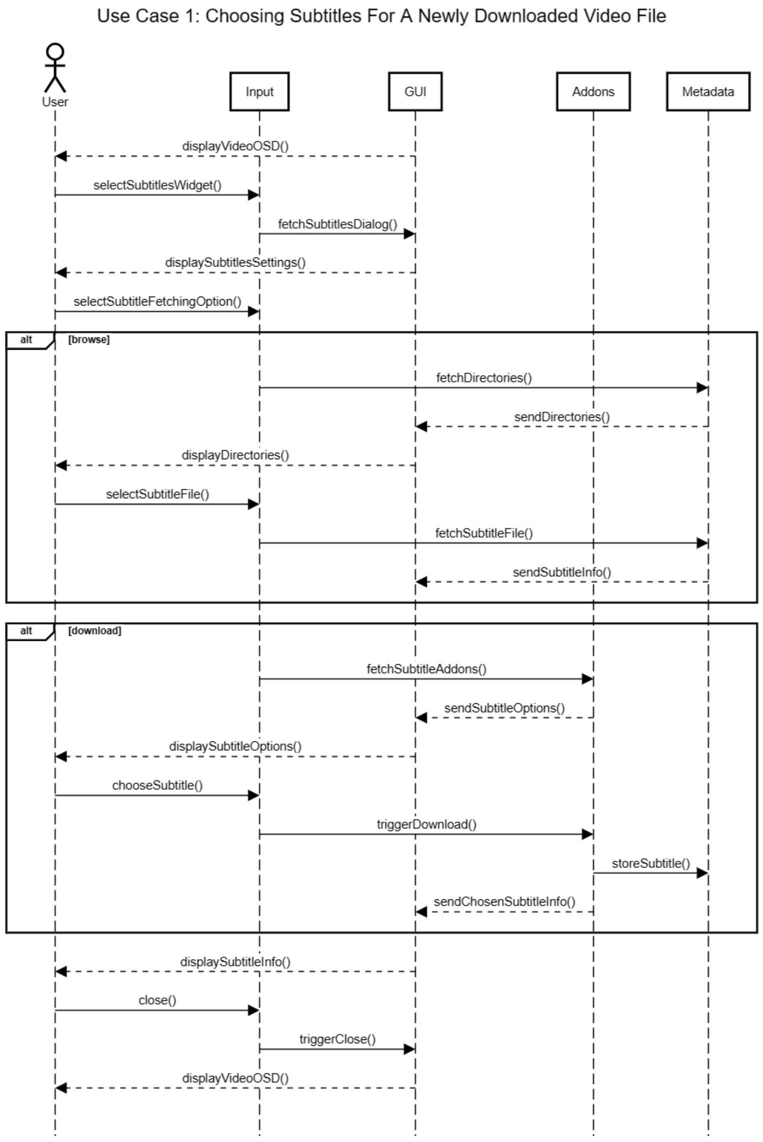

Figure 4: Use Case 1 Sequence Diagram

Use case 2: A user is watching a movie on their computer and would like to create a bookmark so they can save the time code of the section they are on

The user begins in Application.cpp with the video file open and selects the bookmark menu where they are presented with various options. Since they would like to create a new bookmark for the movie they are watching, they select *Create Bookmark*. Once selected, Application.cpp sends details to Bookmark.cpp to create a bookmark at the precise time that the movie is currently at. The new bookmark is sent to Bookmark.h, which checks to ensure that it has been properly set.

If the user has the "Extract thumbnail" option selected, a screenshot is created and saved to be displayed next to the bookmark in the menu of Application.cpp.

Once created, the bookmark is sent to the bookmark database in GUIDialogVideoBookmarks.cpp, where it is added to the list of bookmarks for the current movie. Once added, Application.cpp will update to include the newly added bookmark in the list of existing bookmarks.

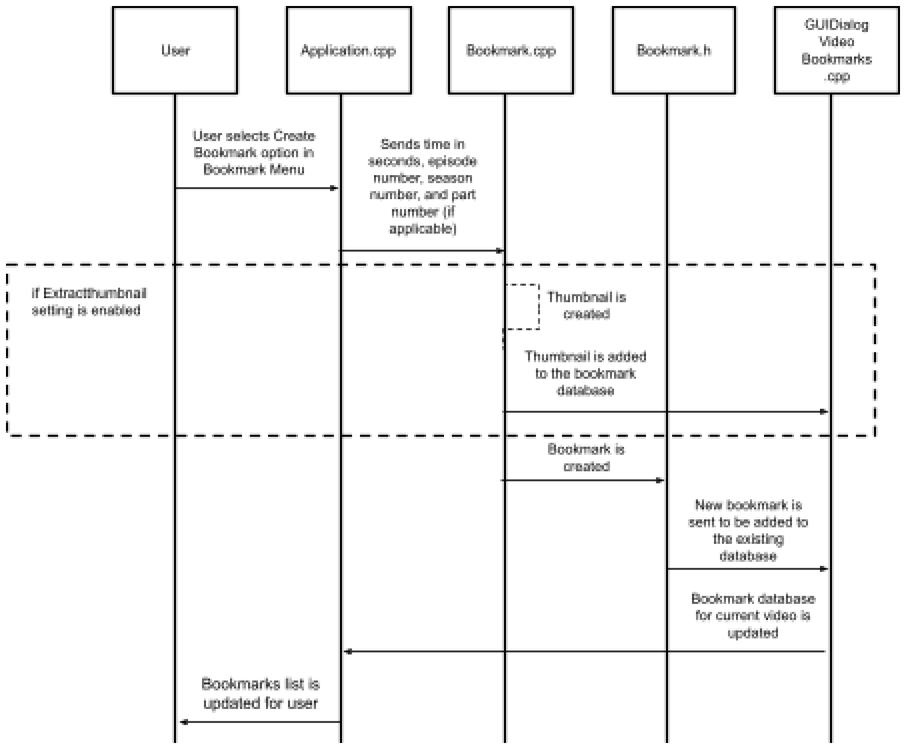
User
Application.cpp
Bookmark.cpp
Bookmark.h
GUIDialog Video Bookmarks .cpp
User selects Create Bookmark option in Bookmark Menu
Sends time in seconds, episode number, season number, and part number (if applicable)
if Extractthumbnail setting is enabled
Thumbnail is created
Thumbnail is added to the bookmark database
Bookmark is created
New bookmark is sent to be added to the existing database
Bookmark database for current video is updated
Bookmarks list is updated for user

Figure 5: Use Case 2 Sequence Diagram

Data Dictionary

Event Server: manages input from devices such as the keyboard, mouse, joystick, etc.
Conceptual Architecture: The architecture of a given software derived without knowledge of the source code.
Concrete Architecture: The derived architecture of software from its source code and its inferred conceptual architecture.
Layered Style Architecture: A type of software architecture that has abstract layers that communicate from top to bottom and vice versa sequentially.
Service Broker: A mechanism for managing queues.
Wrapper: A function that manages a resource. E.g. wrapping a pointer to an int for ease of allocating and deallocating memory (Stack Overflow).

Naming Conventions

Events: Event Server
Services: Web Server
OSD: On-Screen Display
X → Y: X depends on Y
GUI: Graphical User Interface
HDR: High Dynamic Range
HDD: Hard Disk Drive

Lessons Learned

Throughout this report, as a group we faced many challenges and gained an in-depth understanding of the selected Kodi's subsystem and the components involved. The concrete architecture of Kodi follows a layered architecture style, but it deviates from traditional models due to Kodi's open-source nature, leading to unexpected dependencies and interactions between layers. Kodi has demonstrated that not all software programs adhere to strict reference architectures, allowing room for variation and additional components. The architecture comprises four layers: Client Layer, Presentation Layer, Business Layer, and Data Layer, with the Presentation Layer handling user input, GUI management, and player core functionality. The Business Layer includes the Player Core Subsystem and Python Subsystem for add-on management, while the Data Access Layer encompasses the Sources Subsystem and Metadata Subsystem. Reflexion analysis uncovers disparities between the conceptual and concrete architecture, driven by Kodi's open-source nature, resulting in direct database calls for video information and unconventional subtitle handling through add-ons. Use cases illustrate user interactions with the Video Player OSD subsystem, such as creating bookmarks and selecting subtitles, showcasing its functionality and components.

Conclusion

In conclusion, this report has provided an in-depth analysis of Kodi's concrete architecture, shedding light on its layered structure and the intricacies of its subsystems. Our analysis encompassed source code examination from the GitHub repository, dependency tracking using Understand, and consultation with the Kodi Wiki, revealing the system's unique attributes and deviations from traditional models due to its open-source nature. Notable subsystems like the User Input, GUI, and Player Core were explored, while the reflexion analysis highlighted discrepancies between the conceptual and concrete architecture, showcasing the system's adaptability and pragmatic design choices. Lessons learned underscore the challenges and opportunities presented by open-source projects in architectural design. This report serves as an in-depth resource for understanding Kodi's versatile media center software, offering insights into its concrete architecture and the broader implications of open-source development.

References

Kodi Foundation. (n.d.-a). Video playback, Kodi Wiki. https://kodi.wiki/view/Video_playback#Video_Player_OSD

Wikipedia contributors. "Kodi (software)." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 6 Sep. 2023. Web. 19 Nov. 2023. https://en.wikipedia.org/wiki/Kodi_(software)

Kodi development: Dialog. Kodi development. (n.d.). https://alwinesch.github.io/group__python___dialog.html#:~:text=Kodi’s%20dialog%20class,prompts%20them%20for%20a%20response